CPSC 330 Applied Machine Learning

Lecture 20: Survival analysis

UBC 2022-23

Instructor: Mathias Lécuyer

Imports

In [1]:
```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier
from sklearn.ensemble import RandomForestClassifier, RandomForestRegres
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.metrics import confusion_matrix, plot_confusion_matrix
from sklearn.model_selection import (
    cross_val_predict,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import (
    FunctionTransformer,
    OneHotEncoder,
    OrdinalEncoder,
    StandardScaler,
)

plt.rcParams["font.size"] = 16

# does lifelines try to mess with this?
pd.options.display.max_rows = 10
```

In [2]:
```python
import lifelines
```

Learning objectives

- Explain the problem with treating right-censored data the same as "regular" data.

- Determine whether survival analysis is an appropriate tool for a given problem.
- Apply survival analysis in Python using the `lifelines` package.
- Interpret a survival curve, such as the Kaplan-Meier curve.
- Interpret the coefficients of a fitted Cox proportional hazards model.
- Make predictions for existing individuals and interpret these predictions.

Customer churn: our standard approach

- In hw5 you looked at a dataset about [customer churn (https://en.wikipedia.org/wiki/Customer_attrition)](https://en.wikipedia.org/wiki/Customer_attrition).
- In hw5, the dataset was interesting because it's unbalanced (most customers stay). We used typical binary classification approach on the dataset.
- Today we'll look at a different customer churn [dataset (https://www.kaggle.com/blastchar/telco-customer-churn)](https://www.kaggle.com/blastchar/telco-customer-churn), because it has a feature we need - time!
- We'll explore the time aspect of the dataset today.

In [5]:
```
df = pd.read_csv("../data/WA_Fn-UseC_-Telco-Customer-Churn.csv")
train_df, test_df = train_test_split(df, random_state=123)
train_df.head()
```

Out[5]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines
6464	4726-DLWQN	Male	1	No	No	50	Yes	Yes
5707	4537-DKTAL	Female	0	No	No	2	Yes	No
3442	0468-YRPXN	Male	0	No	No	29	Yes	No
3932	1304-NECVQ	Female	1	No	No	2	Yes	Yes
6124	7153-CHRBV	Female	0	Yes	Yes	57	Yes	No

5 rows × 21 columns

We can treat this as a binary classification problem where we want to predict `Churn` (yes/no) from these other columns.

In [6]:
```
train_df.shape
```

Out[6]:
```
(5282, 21)
```

In [7]:
```
train_df["Churn"].value_counts()
```

Out[7]:
```
No     3912
Yes    1370
Name: Churn, dtype: int64
```

In [8]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5282 entries, 6464 to 3582
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        5282 non-null   object 
 1   gender            5282 non-null   object 
 2   SeniorCitizen     5282 non-null   int64  
 3   Partner           5282 non-null   object 
 4   Dependents        5282 non-null   object 
 5   tenure            5282 non-null   int64  
 6   PhoneService      5282 non-null   object 
 7   MultipleLines     5282 non-null   object 
 8   InternetService   5282 non-null   object 
 9   OnlineSecurity    5282 non-null   object 
 10  OnlineBackup      5282 non-null   object 
 11  DeviceProtection  5282 non-null   object 
 12  TechSupport       5282 non-null   object 
 13  StreamingTV       5282 non-null   object 
 14  StreamingMovies   5282 non-null   object 
 15  Contract          5282 non-null   object 
 16  PaperlessBilling  5282 non-null   object 
 17  PaymentMethod     5282 non-null   object 
 18  MonthlyCharges    5282 non-null   float64
 19  TotalCharges      5282 non-null   object 
 20  Churn             5282 non-null   object 
dtypes: float64(1), int64(2), object(18)
memory usage: 907.8+ KB
```

Question: Does this mean there is no missing data?

Ok, let's try our usual approach:

In [9]:
```
train_df["SeniorCitizen"].value_counts()
```

Out[9]:
```
0    4430
1     852
Name: SeniorCitizen, dtype: int64
```

In [10]:
```
numeric_features = ["tenure", "MonthlyCharges", "TotalCharges"]
drop_features = ["customerID"]
passthrough_features = ["SeniorCitizen"]
target_column = ["Churn"]
# the rest are categorical
categorical_features = list(
    set(train_df.columns)
    - set(numeric_features)
    - set(passthrough_features)
    - set(drop_features)
    - set(target_column)
)
```

In [11]:
```
preprocessor = make_column_transformer(
    (StandardScaler(), numeric_features),
    (OneHotEncoder(), categorical_features),
    ("passthrough", passthrough_features),
    ("drop", drop_features),
)
```

In [12]:
```
preprocessor.fit(train_df);
```

```
---------------------------------------------------------------------------
--
ValueError                                Traceback (most recent call las
t)
Cell In[12], line 1
----> 1 preprocessor.fit(train_df);

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/sklearn/com
pose/_column_transformer.py:657, in ColumnTransformer.fit(self, X, y)
    639 """Fit all transformers using X.
    640 
    641 Parameters
   (...)
    653     This estimator.
    654 """
    655 # we use fit_transform to make sure to set sparse_output_ (for wh
ich we
    656 # need the transformed data) to have consistent output type in pr
edict
```

Hmmm, one of the numeric features is causing problems?

In [13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        7043 non-null   object 
 1   gender            7043 non-null   object 
 2   SeniorCitizen     7043 non-null   int64  
 3   Partner           7043 non-null   object 
 4   Dependents        7043 non-null   object 
 5   tenure            7043 non-null   int64  
 6   PhoneService      7043 non-null   object 
 7   MultipleLines     7043 non-null   object 
 8   InternetService   7043 non-null   object 
 9   OnlineSecurity    7043 non-null   object 
 10  OnlineBackup      7043 non-null   object 
 11  DeviceProtection  7043 non-null   object 
 12  TechSupport       7043 non-null   object 
 13  StreamingTV       7043 non-null   object 
 14  StreamingMovies   7043 non-null   object 
 15  Contract          7043 non-null   object 
 16  PaperlessBilling  7043 non-null   object 
 17  PaymentMethod     7043 non-null   object 
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object 
 20  Churn             7043 non-null   object 
dtypes: float64(1), int64(2), object(18)
memory usage: 1.1+ MB
```

Oh, looks like `TotalCharges` is not a numeric type. What if we change the type of this column to float?

In [14]:
```
train_df["TotalCharges"] = train_df["TotalCharges"].astype(float)
```

```
---------------------------------------------------------------------------
--
ValueError                                Traceback (most recent call las
t)
Cell In[14], line 1
----> 1 train_df["TotalCharges"] = train_df["TotalCharges"].astype(float)

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/generic.py:6240, in NDFrame.astype(self, dtype, copy, errors)
   6233     results = [
   6234         self.iloc[:, i].astype(dtype, copy=copy)
   6235         for i in range(len(self.columns))
   6236     ]
   6238 else:
   6239     # else, only a single dtype is given
-> 6240     new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=er
rors)
   6241     return self._constructor(new_data).__finalize__(self, method
="astype")
   6243 # GH 33113: handle empty frame or series

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/internals/managers.py:448, in BaseBlockManager.astype(self, dtype, cop
y, errors)
    447 def astype(self: T, dtype, copy: bool = False, errors: str = "rai
se") -> T:
--> 448     return self.apply("astype", dtype=dtype, copy=copy, errors=er
rors)

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/internals/managers.py:352, in BaseBlockManager.apply(self, f, align_key
s, ignore_failures, **kwargs)
    350         applied = b.apply(f, **kwargs)
    351     else:
--> 352         applied = getattr(b, f)(**kwargs)
    353 except (TypeError, NotImplementedError):
    354     if not ignore_failures:

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/internals/blocks.py:526, in Block.astype(self, dtype, copy, errors)
    508 """
    509 Coerce to the new dtype.
    510
   (...)
    522 Block
    523 """
    524 values = self.values
--> 526 new_values = astype_array_safe(values, dtype, copy=copy, errors=e
rrors)
    528 new_values = maybe_coerce_values(new_values)
    529 newb = self.make_block(new_values)

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/dtypes/astype.py:299, in astype_array_safe(values, dtype, copy, errors)
    296     return values.copy()
    298 try:
--> 299     new_values = astype_array(values, dtype, copy=copy)
```

```
    300 except (ValueError, TypeError):
    301     # e.g. astype_nansafe can fail on object-dtype of strings
    302     #  trying to convert to float
    303     if errors == "ignore":

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/dtypes/astype.py:230, in astype_array(values, dtype, copy)
    227     values = values.astype(dtype, copy=copy)
    229 else:
--> 230     values = astype_nansafe(values, dtype, copy=copy)
    232 # in pandas we don't store numpy str dtypes, so convert to object
    233 if isinstance(dtype, np.dtype) and issubclass(values.dtype.type,
str):

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/pandas/cor
e/dtypes/astype.py:170, in astype_nansafe(arr, dtype, copy, skipna)
    166     raise ValueError(msg)
    168 if copy or is_object_dtype(arr.dtype) or is_object_dtype(dtype):
    169     # Explicit copy, or required since NumPy can't view from / to
object.
--> 170     return arr.astype(dtype, copy=True)
    172 return arr.astype(dtype, copy=copy)

ValueError: could not convert string to float: ' '
```

Argh!!

In [15]:

```python
for val in train_df["TotalCharges"]:
    try:
        float(val)
    except ValueError:
        print(val)
```

Any ideas?

Well, it turns out we can't see those problematic values because they are whitespace!

In [16]:
```
for val in train_df["TotalCharges"]:
    try:
        float(val)
    except ValueError:
        print('"%s"' % val)
```

```
" "
" "
" "
" "
" "
" "
" "
" "
```

Let's replace the whitespaces with NaNs.

In [17]:
```
train_df = train_df.assign(
    TotalCharges=train_df["TotalCharges"].replace(" ", np.nan).astype(f
)
test_df = test_df.assign(
    TotalCharges=test_df["TotalCharges"].replace(" ", np.nan).astype(fl
)
```

In [18]:
```
train_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 5282 entries, 6464 to 3582
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   customerID        5282 non-null   object 
 1   gender            5282 non-null   object 
 2   SeniorCitizen     5282 non-null   int64  
 3   Partner           5282 non-null   object 
 4   Dependents        5282 non-null   object 
 5   tenure            5282 non-null   int64  
 6   PhoneService      5282 non-null   object 
 7   MultipleLines     5282 non-null   object 
 8   InternetService   5282 non-null   object 
 9   OnlineSecurity    5282 non-null   object 
 10  OnlineBackup      5282 non-null   object 
 11  DeviceProtection  5282 non-null   object 
 12  TechSupport       5282 non-null   object 
 13  StreamingTV       5282 non-null   object 
 14  StreamingMovies   5282 non-null   object 
 15  Contract          5282 non-null   object 
 16  PaperlessBilling  5282 non-null   object 
 17  PaymentMethod     5282 non-null   object 
 18  MonthlyCharges    5282 non-null   float64
 19  TotalCharges      5274 non-null   float64
 20  Churn             5282 non-null   object 
dtypes: float64(2), int64(2), object(17)
memory usage: 907.8+ KB
```

But now we are going to have missing values and we need to include imputation for numeric features in our preprocessor.

In [19]:
```
preprocessor = make_column_transformer(
    (
        make_pipeline(SimpleImputer(strategy="median"), StandardScaler(
        numeric_features,
    ),
    (OneHotEncoder(handle_unknown="ignore"), categorical_features),
    ("passthrough", passthrough_features),
    ("drop", drop_features),
)
```

Now let's try that again...

In [20]:
```
preprocessor.fit(train_df);
```

It worked! Let's get the column names of the transformed data from the column transformer.

In [22]:
```
new_columns = (
    numeric_features
    + preprocessor.named_transformers_["onehotencoder"]
    .get_feature_names_out(categorical_features)
    .tolist()
    + passthrough_features
)
```

In [23]:
```
X_train_enc = pd.DataFrame(
    preprocessor.transform(train_df), index=train_df.index, columns=new
)
X_test_enc = pd.DataFrame(
    preprocessor.transform(train_df), index=train_df.index, columns=new
)
```

In [24]:
```
X_train_enc.head()
```

Out[24]:

	tenure	MonthlyCharges	TotalCharges	TechSupport_No	TechSupport_No internet service	TechSupport_Yes
6464	0.707712	0.185175	0.513678	1.0	0.0	0.0
5707	-1.248999	-0.641538	-0.979562	1.0	0.0	0.0
3442	-0.148349	1.133562	0.226789	0.0	0.0	1.0
3932	-1.248999	0.458524	-0.950696	1.0	0.0	0.0
6124	0.993065	-0.183179	0.433814	0.0	0.0	1.0

5 rows × 45 columns

Before we look into survival analysis, let's just treat it as a binary classification model where we want to predict whether a customer churned or not.

In [25]:
```
results = {}
```

In [26]:
```
def mean_std_cross_val_scores(model, X_train, y_train, **kwargs):
    """
    Returns mean and std of cross validation

    Parameters
    ----------
    model :
        scikit-learn model
    X_train : numpy array or pandas DataFrame
        X in the training data
    y_train :
        y in the training data

    Returns
    ----------
        pandas Series with mean scores from cross_validation
    """

    scores = cross_validate(model, X_train, y_train, **kwargs)

    mean_scores = pd.DataFrame(scores).mean()
    std_scores = pd.DataFrame(scores).std()
    out_col = []

    for i in range(len(mean_scores)):
        out_col.append((f"%0.3f (+/- %0.3f)" % (mean_scores[i], std_sco

    return pd.Series(data=out_col, index=mean_scores.index)
```

In [27]:
```
X_train = train_df.drop(columns=["Churn"])
X_test = test_df.drop(columns=["Churn"])

y_train = train_df["Churn"]
y_test = test_df["Churn"]
```

DummyClassifier

In [28]:
```
dc = DummyClassifier()
```

In [29]:
```
results["dummy"] = mean_std_cross_val_scores(
    dc, X_train, y_train, return_train_score=True
)
pd.DataFrame(results)
```

Out[29]:

	dummy
fit_time	0.003 (+/- 0.001)
score_time	0.001 (+/- 0.000)
test_score	0.741 (+/- 0.000)
train_score	0.741 (+/- 0.000)

LogisticRegression

In [30]:
```
lr = make_pipeline(preprocessor, LogisticRegression(max_iter=1000))
```

In [31]:
```
results["logistic regression"] = mean_std_cross_val_scores(
    lr, X_train, y_train, return_train_score=True
)
pd.DataFrame(results)
```

Out[31]:

	dummy	logistic regression
fit_time	0.003 (+/- 0.001)	0.055 (+/- 0.008)
score_time	0.001 (+/- 0.000)	0.006 (+/- 0.000)
test_score	0.741 (+/- 0.000)	0.804 (+/- 0.013)
train_score	0.741 (+/- 0.000)	0.809 (+/- 0.002)

In [32]:
```
confusion_matrix(y_train, cross_val_predict(lr, X_train, y_train))
```

Out[32]:
```
array([[3516,  396],
       [ 637,  733]])
```

RandomForestClassifier

In [33]:
```
rf = make_pipeline(preprocessor, RandomForestClassifier())
```

In [34]:
```
results["random forest"] = mean_std_cross_val_scores(
    rf, X_train, y_train, return_train_score=True
)
pd.DataFrame(results)
```

Out[34]:

	dummy	logistic regression	random forest
fit_time	0.003 (+/- 0.001)	0.055 (+/- 0.008)	0.287 (+/- 0.014)
score_time	0.001 (+/- 0.000)	0.006 (+/- 0.000)	0.020 (+/- 0.001)
test_score	0.741 (+/- 0.000)	0.804 (+/- 0.013)	0.790 (+/- 0.012)
train_score	0.741 (+/- 0.000)	0.809 (+/- 0.002)	0.998 (+/- 0.000)

In [35]:
```
confusion_matrix(y_train, cross_val_predict(rf, X_train, y_train))
```

Out[35]:
```
array([[3518,  394],
       [ 735,  635]])
```

- This is was we did in hw5.
- What's wrong with this approach?

And now the rest of the class is about what is wrong with what we just did!

Censoring and survival analysis

Time to event and censoring

Imagine that you want to analyze *the time until an event occurs*. For example,

- the time until a disease kills its host.
- the time until a piece of equipment breaks.
- the time that someone unemployed will take to land a new job.
- the time until a customer leaves a subscription service (this dataset).

In our example, instead of predicting the binary label churn or no churn, it will be more useful to estimate when the customer is likely to churn (the time until churn happens) so that we can take some action.

In [36]:
```
train_df[["tenure"]].head()
```

Out[36]:

	tenure
6464	50
5707	2
3442	29
3932	2
6124	57

The tenure column is the number of months the customer has stayed with the company.

Although this branch of statistics is usually referred to as **Survival Analysis**, the event in question does not need to be related to actual "survival". The important thing is to understand that we are interested in **the time until something happens**, or whether or not something will happen in a certain time frame.

Question: But why is this different? Can't you just use the techniques you learned so far (e.g., regression models) to predict the time? Take a minute to think about this.

The answer would be yes if you could observe the actual time in all occurrences, but you usually cannot. Frequently, there will be some kind of **censoring** which will not allow you to observe the exact time that the event happened for all units/individuals that are being studied.

In [37]:
```
train_df[["tenure", "Churn"]].head()
```

Out[37]:

	tenure	Churn
6464	50	No
5707	2	No
3442	29	No
3932	2	Yes
6124	57	No

- What this means is that we **don't have correct target values** to train or test our model.
- This is a problem!

Let's consider some approaches to deal with this censoring issue.

Approach 1: Only consider the examples where "Churn"=Yes

Let's just consider the cases *for which we have the time*, to obtain the average subscription length.

In [38]:

```
train_df_churn = train_df.query(
    "Churn == 'Yes'"
)  # Consider only examples where the customers churned.
test_df_churn = test_df.query(
    "Churn == 'Yes'"
)  # Consider only examples where the customers churned.
train_df_churn.head()
```

Out[38]:

	customerID	gender	SeniorCitizen	Partner	Dependents	tenure	PhoneService	MultipleLines
3932	1304-NECVQ	Female	1	No	No	2	Yes	Yes
301	8098-LLAZX	Female	1	No	No	4	Yes	Yes
5540	3803-KMQFW	Female	0	Yes	Yes	1	Yes	No
4084	2777-PHDEI	Female	0	No	No	1	Yes	No
3272	6772-KSATR	Male	0	No	No	1	Yes	Yes

5 rows × 21 columns

In [39]:

```
train_df.shape
```

Out[39]: (5282, 21)

In [40]:

```
train_df_churn.shape
```

Out[40]: (1370, 21)

In [41]:

```
numeric_features
```

Out[41]: ['tenure', 'MonthlyCharges', 'TotalCharges']

In [42]:

```
preprocessing_notenure = make_column_transformer(
    (
        make_pipeline(SimpleImputer(strategy="median"), StandardScaler(
        numeric_features[1:],  # Getting rid of the tenure column
    ),
    (OneHotEncoder(handle_unknown="ignore"), categorical_features),
    ("passthrough", passthrough_features),
)
```

In [43]:

```
tenure_lm = make_pipeline(preprocessing_notenure, Ridge())

tenure_lm.fit(train_df_churn.drop(columns=["tenure"]), train_df_churn["
```

In [44]:
```
pd.DataFrame(
    tenure_lm.predict(test_df_churn.drop(columns=["tenure"]))[:10],
    columns=["tenure_predictions"],
)
```

Out[44]:

	tenure_predictions
0	5.062449
1	13.198645
2	11.859455
3	5.865562
4	58.154842
5	3.757932
6	18.932070
7	7.720893
8	36.818041
9	7.263541

What will be wrong with our estimated survival times? Will they be too low or too high?

On average they will be **underestimates** (too small), because we are ignoring the currently subscribed (un-churned) customers. Our dataset is a biased sample of those who churned within the time window of the data collection. Long-time subscribers were more likely to be removed from the dataset! This is a common mistake - see the [Calling Bullshit video (https://www.youtube.com/watch?v=ITWQ5psx9Sw)](https://www.youtube.com/watch?v=ITWQ5psx9Sw) I posted on the README!

Approach 2: Assume everyone churns right now

Assume everyone churns right now - in other words, use the original dataset.

In [45]:

```
train_df[["tenure", "Churn"]].head()
```

Out[45]:

	tenure	Churn
6464	50	No
5707	2	No
3442	29	No
3932	2	Yes
6124	57	No

In [46]:

```
tenure_lm.fit(train_df.drop(columns=["tenure"]), train_df["tenure"]);
```

In [47]:

```
pd.DataFrame(
    tenure_lm.predict(test_df_churn.drop(columns=["tenure"]))[:10],
    columns=["tenure_predictions"],
)
```

Out[47]:

	tenure_predictions
0	6.400047
1	20.220392
2	22.332746
3	12.825470
4	59.885968
5	7.075453
6	17.731498
7	10.407862
8	38.425365
9	10.854500

What will be wrong with our estimated survival time?

In [48]:
```
train_df[["tenure", "Churn"]].head()
```

Out[48]:

	tenure	Churn
6464	50	No
5707	2	No
3442	29	No
3932	2	Yes
6124	57	No

It will be an **underestimate** again. For those still subscribed, while we did not remove them, we recorded a total tenure shorter than in reality, because they will keep going for some amount of time.

Approach 3: Survival analysis

Deal with this properly using [survival analysis (https://en.wikipedia.org/wiki/Survival_analysis)](https://en.wikipedia.org/wiki/Survival_analysis).

- You may learn about this in a statistics course.
- We will use the `lifelines` package in Python and will not go into the math/stats of how it works.

In [49]:
```
train_df[["tenure", "Churn"]].head()
```

Out[49]:

	tenure	Churn
6464	50	No
5707	2	No
3442	29	No
3932	2	Yes
6124	57	No

Types of questions we might want to answer:

1. How long do customers stay with the service?
2. For a particular customer, can we predict how long they might stay with the service?
3. What factors influence a customer's churn time?

Break (5 min)

Kaplan-Meier survival curve

Before we do anything further, I want to modify our dataset slightly:

1. I'm going to drop the `TotalCharges` (yes, after all that work fixing it) because it's a bit of a strange feature.

- Its value actually changes over time, but we only have the value at the end.
- We still have `MonthlyCharges` .

2. I'm going to not scale the `tenure` column, since it will be convenient to keep it in its original units of months.

Just for our sanity, I'm redefining the features.

In [50]:
```
numeric_features = ["MonthlyCharges"]
drop_features = ["customerID", "TotalCharges"]
passthrough_features = ["tenure", "SeniorCitizen"]  # don't want to sca
target_column = ["Churn"]
# the rest are categorical
categorical_features = list(
    set(train_df.columns)
    - set(numeric_features)
    - set(passthrough_features)
    - set(drop_features)
    - set(target_column)
)
```

In [51]:
```
preprocessing_final = make_column_transformer(
    (
        FunctionTransformer(lambda x: x == "Yes"),
        target_column,
    ),  # because we need it in this format for lifelines package
    ("passthrough", passthrough_features),
    (StandardScaler(), numeric_features),
    (OneHotEncoder(handle_unknown="ignore", sparse=False), categorical_
    ("drop", drop_features),
)
```

In [52]:
```
preprocessing_final.fit(train_df);
```

Let's get the column names of the columns created by our column transformer.

In [54]:
```
new_columns = (
    target_column
    + passthrough_features
    + numeric_features
    + preprocessing_final.named_transformers_["onehotencoder"]
    .get_feature_names_out(categorical_features)
    .tolist()
)
```

In [55]:
```
train_df_surv = pd.DataFrame(
    preprocessing_final.transform(train_df), index=train_df.index, colu
)
test_df_surv = pd.DataFrame(
    preprocessing_final.transform(test_df), index=test_df.index, column
)
```

In [56]:
```
train_df_surv.head()
```

Out[56]:

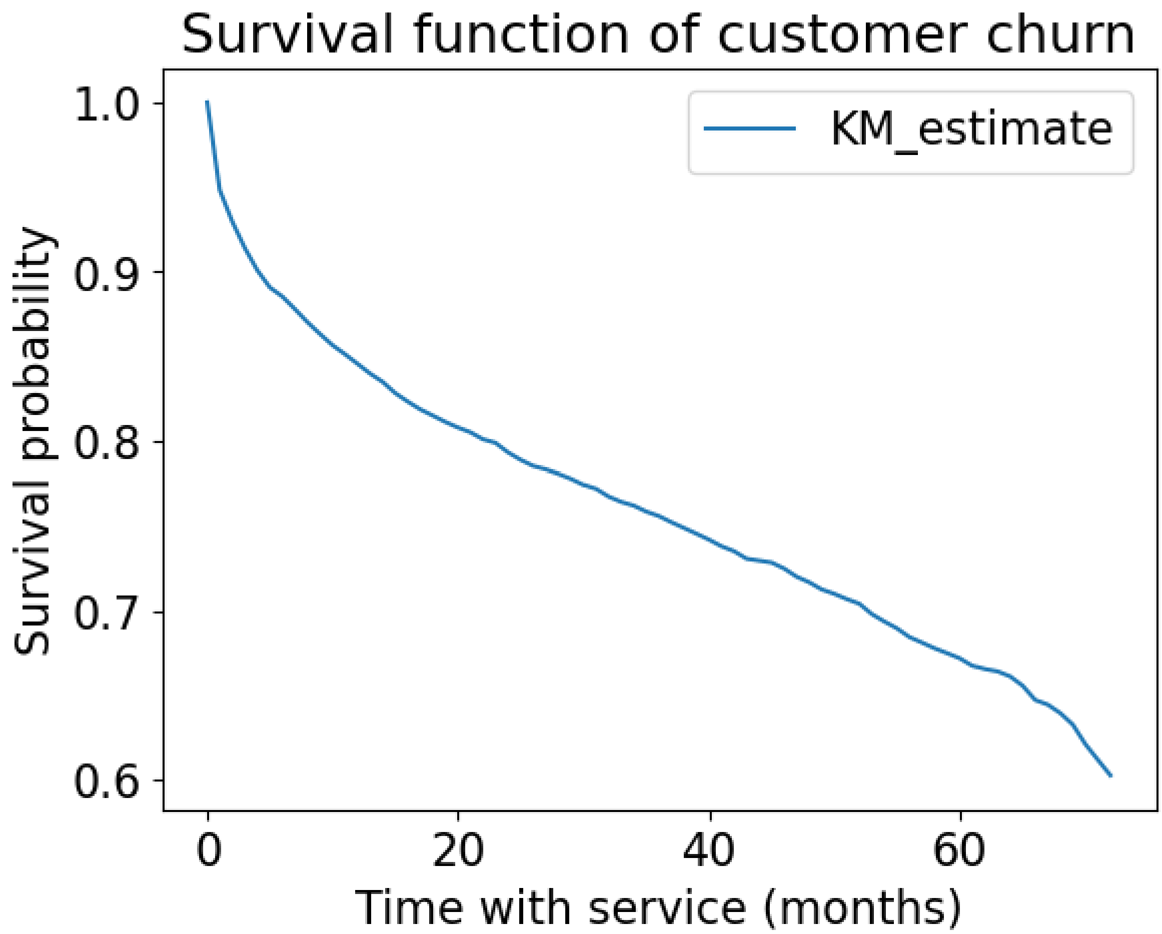
	Churn	tenure	SeniorCitizen	MonthlyCharges	TechSupport_No	TechSupport_No internet service	TechSupport_
6464	0.0	50.0	1.0	0.185175	1.0	0.0	
5707	0.0	2.0	0.0	-0.641538	1.0	0.0	
3442	0.0	29.0	0.0	1.133562	0.0	0.0	
3932	1.0	2.0	1.0	0.458524	1.0	0.0	
6124	0.0	57.0	0.0	-0.183179	0.0	0.0	

5 rows × 45 columns

- We'll start with a model called `KaplanMeierFitter` from `lifelines` package to get a Kaplan Meier curve.
- For this model we only use two columns: tenure and churn.
- We do not use any other features.

In [57]:
```
kmf = lifelines.KaplanMeierFitter()
kmf.fit(train_df_surv["tenure"], train_df_surv["Churn"]);
```

In [58]:
```
kmf.survival_function_.plot()
plt.title("Survival function of customer churn")
plt.xlabel("Time with service (months)")
plt.ylabel("Survival probability");
```



- What is this plot telling us?
- It shows the probability of survival over time.
- For example, after 20 months the probability of survival is ~0.8.
- Over time it's going down.

In [64]:
```
np.mean(y_train == "No")
```

Out[64]: 0.7406285497917455

What's the average tenure?

In [65]:
```
np.mean(train_df_surv["tenure"])
```

Out[65]: 32.6391518364256

What's the average tenure of the people who churned?

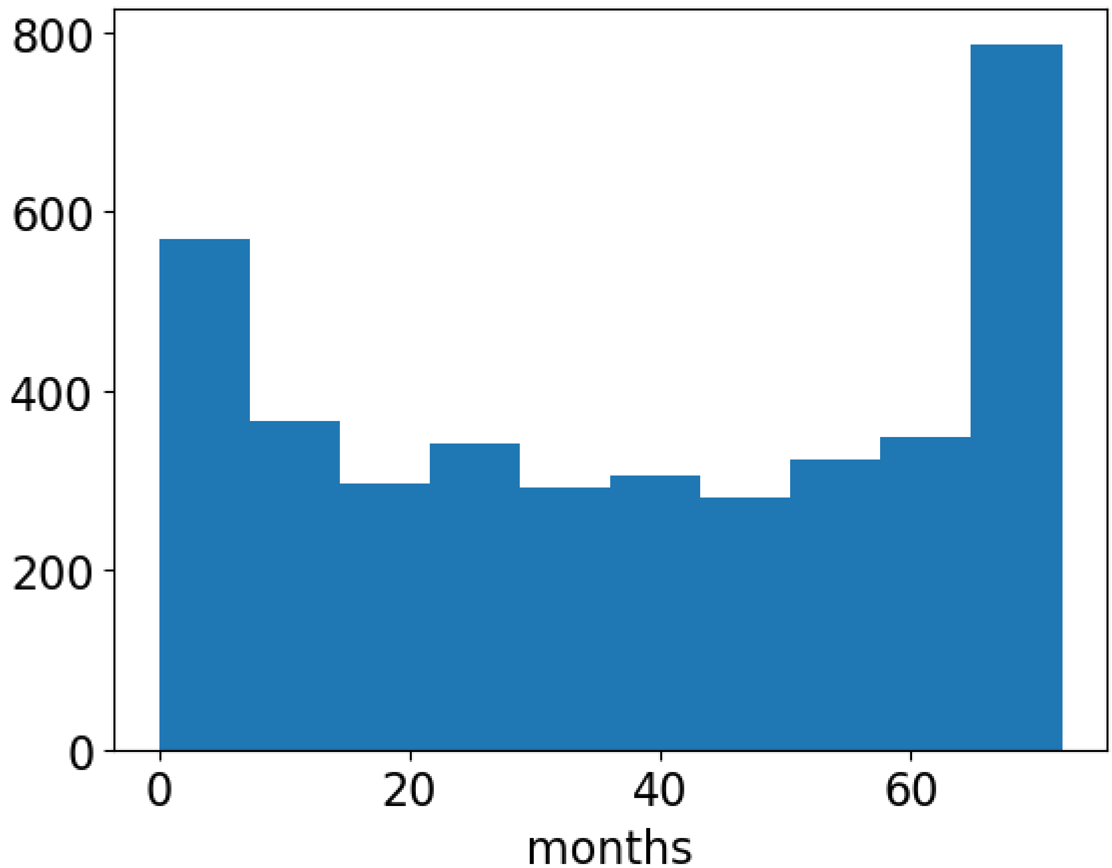
In [66]:
```
np.mean(train_df_surv.query("Churn == 1.0")["tenure"])
```
Out[66]: 17.854744525547446

What's the average tenure of the people who did not churn?

In [67]:
```
np.mean(train_df_surv.query("Churn == 0.0")["tenure"])
```
Out[67]: 37.816717791411044

- Let's look at the histogram of number of people who have not churned.
- The key point here is that people *joined at different times*.

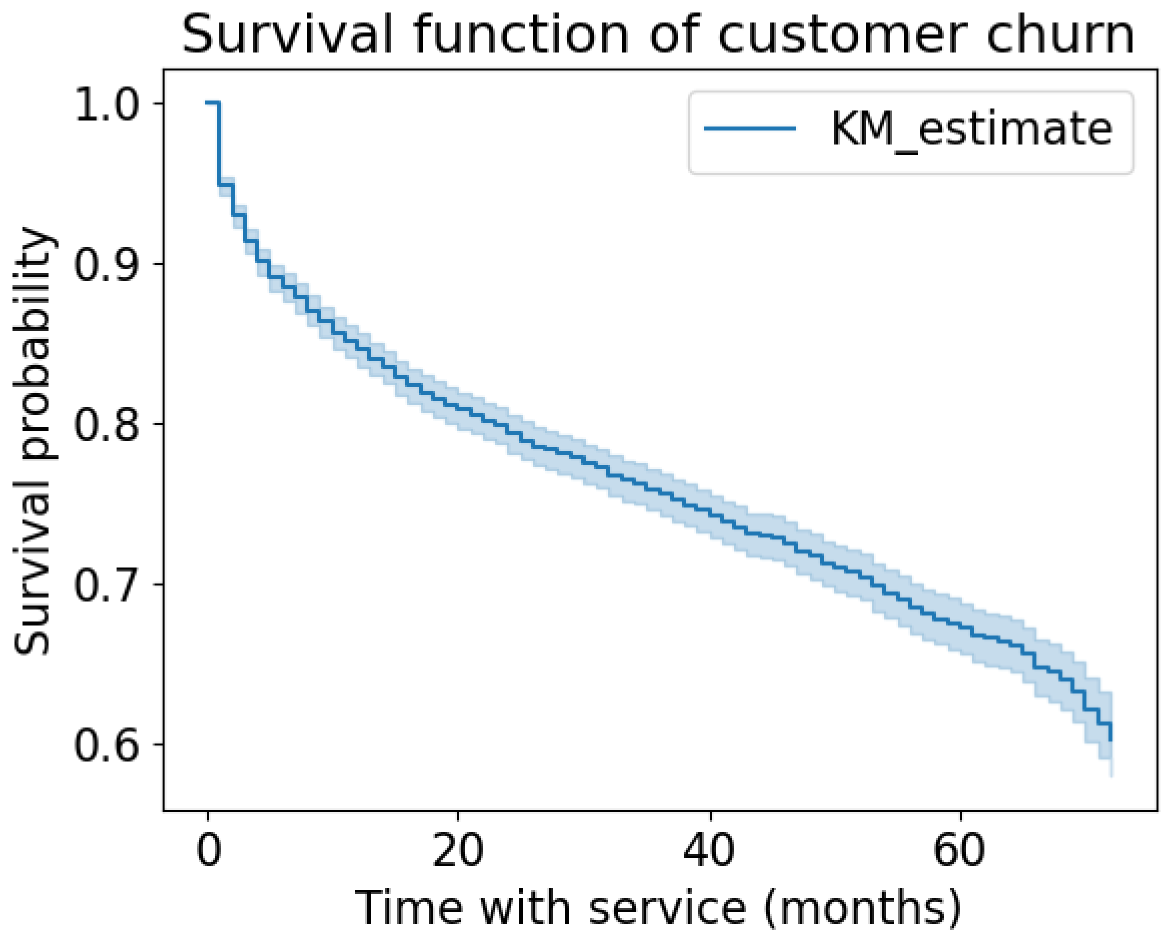
In [68]:
```
train_df[y_train == "No"]["tenure"].hist(grid=False)
plt.xlabel("months");
```



- Since the data was collected at a fixed time and these are the people who hadn't yet churned, those with larger `tenure` values here must have joined earlier.

Lifelines can also give us some "error bars":

In [69]:
```
kmf.plot()
plt.title("Survival function of customer churn")
plt.xlabel("Time with service (months)")
plt.ylabel("Survival probability");
```



- We already have some actionable information here.
- The curve drops down fast at the beginning suggesting that people tend to leave early on.
- If there would have been a big drop in the curve, it means a bunch of people left at that time (e.g., after a 1-month free trial).
- BTW, the [original paper by Kaplan and Meier (https://web.stanford.edu/~lutian/coursepdf/KMpaper.pdf)](https://web.stanford.edu/~lutian/coursepdf/KMpaper.pdf) has been cited over 57000 times!

We can also create the K-M curve for different subgroups:

In [70]:
```
T = train_df_surv["tenure"]
E = train_df_surv["Churn"]
senior = train_df_surv["SeniorCitizen"] == 1
```

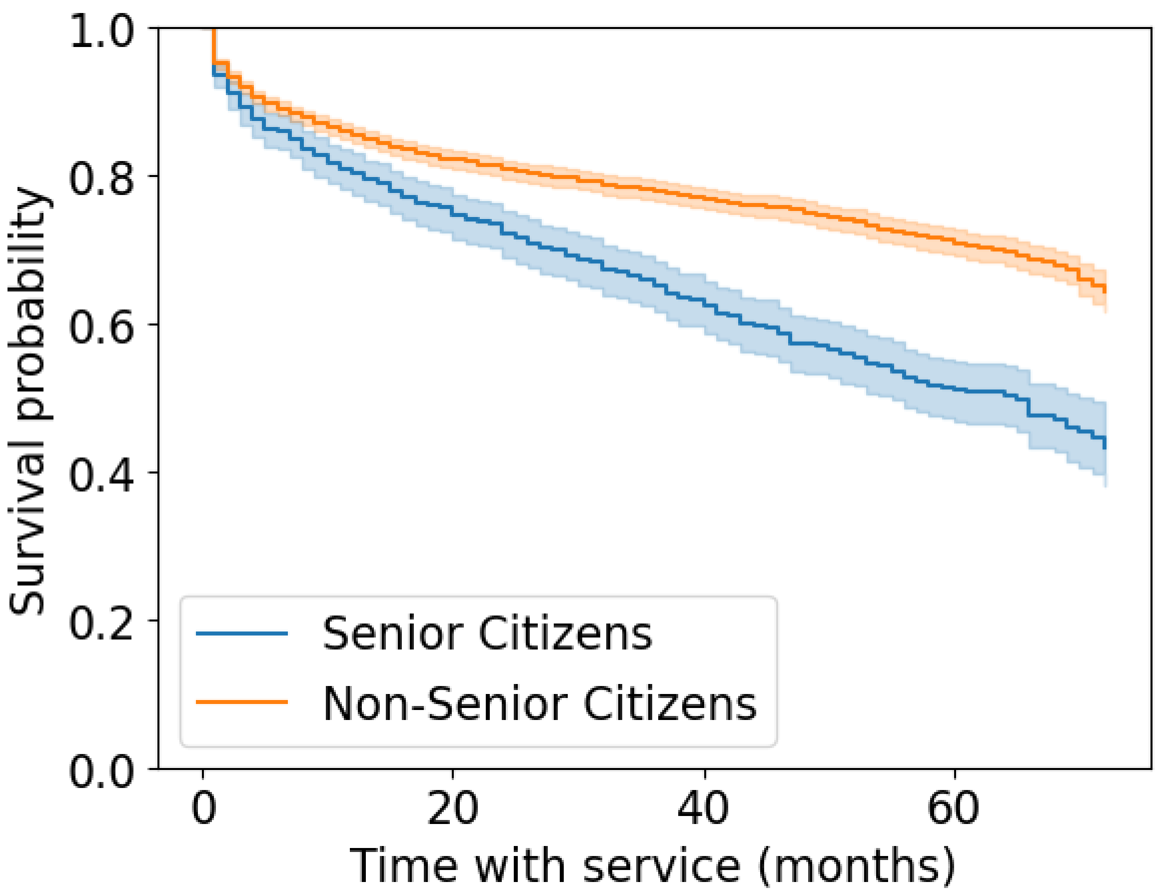
In [71]:

```
ax = plt.subplot(111)

kmf.fit(T[senior], event_observed=E[senior], label="Senior Citizens")
kmf.plot(ax=ax)

kmf.fit(T[~senior], event_observed=E[~senior], label="Non-Senior Citize
kmf.plot(ax=ax)

plt.ylim(0, 1)
plt.xlabel("Time with service (months)")
plt.ylabel("Survival probability");
```



- It looks like senior citizens churn more quickly than others.
- This is quite useful!

Cox proportional hazards model

- We haven't been incorporating other features in the model so far.
- The Cox proportional hazards model is a commonly used model that allows us to interpret how features influence a censored tenure/duration.

- You can think of it like linear regression for survival analysis: we will get a coefficient for each feature that tells us how it influences survival.
- It makes some strong assumptions (the proportional hazards assumption) that may not be true, but we won't go into this here.
- The proportional hazard model works multiplicatively, like linear regression with log-transformed targets.

In [72]:
```
cph = lifelines.CoxPHFitter()
cph.fit(train_df_surv, duration_col="tenure", event_col="Churn");
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/lifelines/fitters/coxph_fitter.py:1527, in SemiParametricPHFitter._newton_raphson_for_efron_model(self, X, T, E, weights, entries, initial_point, show_progress, step_size, precision, max_steps)
   1526 try:
-> 1527     inv_h_dot_g_T = spsolve(-h, g, assume_a="pos", check_finite=False)
   1528 except (ValueError, LinAlgError) as e:

File /opt/anaconda3/envs/cpsc330/lib/python3.10/site-packages/scipy/linalg/_basic.py:254, in solve(a, b, sym_pos, lower, overwrite_a, overwrite_b, check_finite, assume_a, transposed)
    251 lu, x, info = posv(a1, b1, lower=lower,
    252                    overwrite_a=overwrite_a,
    253                    overwrite_b=overwrite_b)
--> 254 _solve_check(n, info)
```

- Ok, going that [that URL (https://lifelines.readthedocs.io/en/latest/Examples.html#problems-with-convergence-in-the-cox-proportional-hazard-model)](https://lifelines.readthedocs.io/en/latest/Examples.html#problems-with-convergence-in-the-cox-proportional-hazard-model), it seems the easiest solution is to add a penalizer.
  - FYI this is related to switching from `LinearRegression` to `Ridge`.
  - Adding `drop='first'` on our OHE might have helped with this.
  - (For 340 folks: we're adding regularization; `lifelines` adds both L1 and L2 regularization, aka elastic net)

In [73]:
```
cph = lifelines.CoxPHFitter(penalizer=0.1)
cph.fit(train_df_surv, duration_col="tenure", event_col="Churn");
```

We can look at the coefficients learned by the model and start interpreting them!

In [74]:
```
cph_params = pd.DataFrame(cph.params_).sort_values(by="coef", ascending
cph_params
```

Out[74]:

	coef
covariate	
Contract_Month-to-month	0.812875
OnlineSecurity_No	0.311151
OnlineBackup_No	0.298561
PaymentMethod_Electronic check	0.280801
Partner_No	0.244814
...	...
OnlineBackup_Yes	-0.282600
PaymentMethod_Credit card (automatic)	-0.302801
OnlineSecurity_Yes	-0.330346
Contract_One year	-0.351821
Contract_Two year	-0.776427

43 rows × 1 columns

- Looks like month-to-month leads to more churn, two-year contract leads to less churn; this makes sense!!!

In [80]:
```
# cph.baseline_hazard_ # baseline hazard
```

In [79]:
```
# cph.summary
```

Could we have gotten this type of information out of sklearn?

In [77]:
```
y_train.head()
```

Out[77]:
```
6464     No
5707     No
3442     No
3932    Yes
6124     No
Name: Churn, dtype: object
```

In [78]:
```
X_train.drop(columns=["tenure"]).head()
```

Out[78]:

	customerID	gender	SeniorCitizen	Partner	Dependents	PhoneService	MultipleLines	Internet
6464	4726-DLWQN	Male	1	No	No	Yes	Yes	
5707	4537-DKTAL	Female	0	No	No	Yes	No	
3442	0468-YRPXN	Male	0	No	No	Yes	No	Fit
3932	1304-NECVQ	Female	1	No	No	Yes	Yes	Fit
6124	7153-CHRBV	Female	0	Yes	Yes	Yes	No	

I'm redefining feature types and our preprocessor for our sanity.

In [81]:
```
numeric_features = ["MonthlyCharges", "TotalCharges"]
drop_features = ["customerID", "tenure"]
passthrough_features = ["SeniorCitizen"]
target_column = ["Churn"]
# the rest are categorical
categorical_features = list(
    set(train_df.columns)
    - set(numeric_features)
    - set(passthrough_features)
    - set(drop_features)
    - set(target_column)
)
```

In [82]:
```
preprocessor = make_column_transformer(
    (
        make_pipeline(SimpleImputer(strategy="median"), StandardScaler(
        numeric_features,
    ),
    (OneHotEncoder(handle_unknown="ignore"), categorical_features),
    ("passthrough", passthrough_features),
    ("drop", drop_features),
)
```

In [83]:
```
preprocessor.fit(X_train);
```

In [86]:
```
new_columns = (
    numeric_features
    + preprocessor.named_transformers_["onehotencoder"]
    .get_feature_names_out(categorical_features)
    .tolist()
    + passthrough_features
)
```

In [87]:

```
lr = make_pipeline(preprocessor, LogisticRegression(max_iter=1000))
lr.fit(X_train, y_train)
lr_coefs = pd.DataFrame(
    data=np.squeeze(lr[1].coef_), index=new_columns, columns=["Coeffici
)
```

In [88]:

```
lr_coefs.sort_values(by="Coefficient", ascending=False)
```

Out[88]:

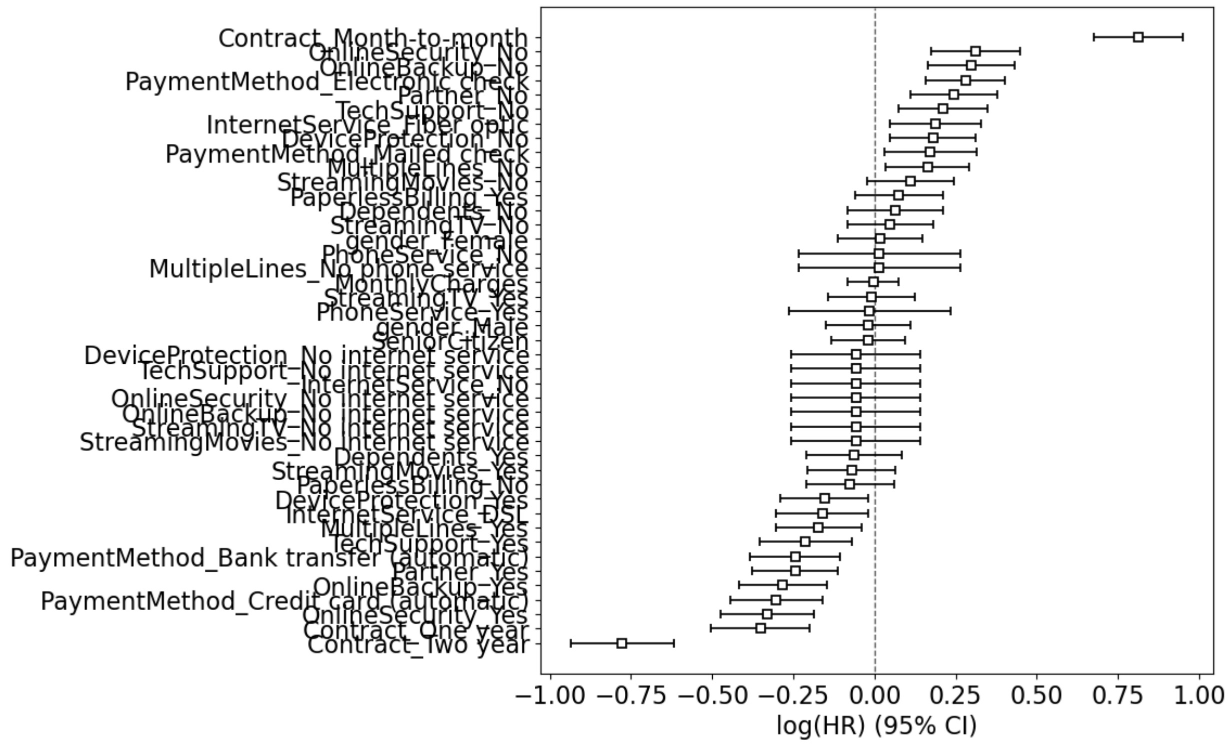
	Coefficient
Contract_Month-to-month	0.787653
InternetService_Fiber optic	0.600509
OnlineSecurity_No	0.291008
StreamingTV_Yes	0.258659
PaymentMethod_Electronic check	0.251646
...	...
MultipleLines_No	-0.169654
PaymentMethod_Credit card (automatic)	-0.204406
InternetService_DSL	-0.461593
TotalCharges	-0.743315
Contract_Two year	-0.765519

44 rows × 1 columns

- There is some agreement, which is good.
- But our survival model is much more useful.
  - Not to mention more correct.

- One thing we get with `lifelines` is confidence intervals on the coefficients:

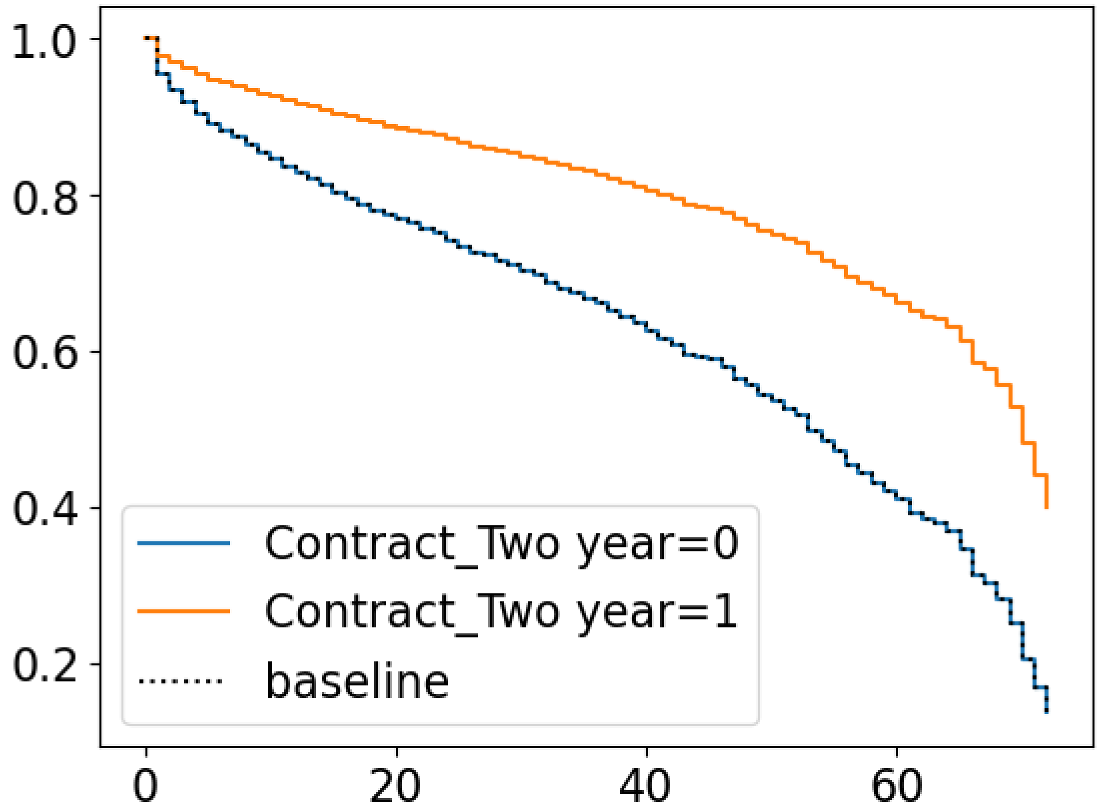
In [89]:
```
plt.figure(figsize=(8, 8))
cph.plot();
```



- (We could probably get the same for logistic regression if using `statsmodels` instead of sklearn.)
- However, in general, I would be careful with all of this.
- Ideally we would have more statistical training when using `lifelines` - there is a lot that can go wrong.
  - It comes with various diagnostics as well.
- But I think it's very useful to know about survival analysis and the availability of software to deal with it.
- Oh, and there are lots of other nice plots.

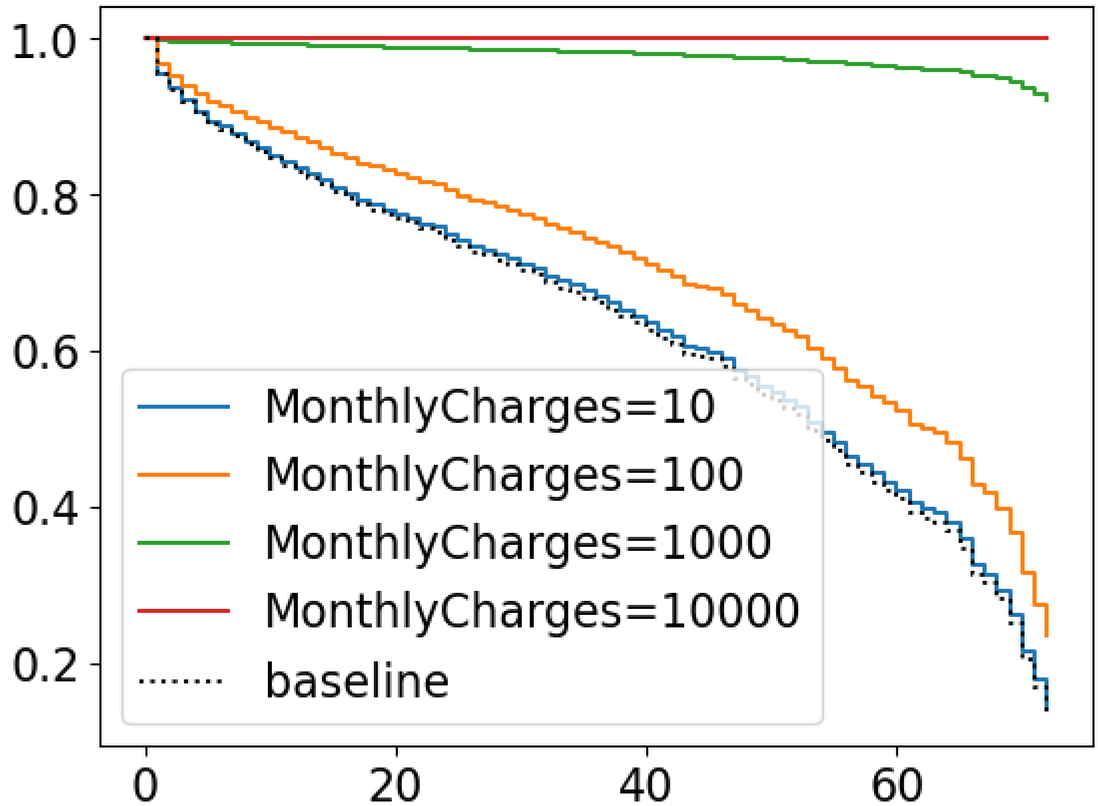
- Let's look at the survival plots for the people with
  - two-year contract (Contract_Two year = 1) and
  - people without two-year contract (Contract_Two year = 0)
- As expected, the former survive longer.

In [90]:
```
cph.plot_partial_effects_on_outcome("Contract_Two year", [0, 1]);
```



Now let's look at the survival plots for the people with different MonthlyCharges.

In [91]:
```
cph.plot_partial_effects_on_outcome("MonthlyCharges", [10, 100, 1000, 1
```



- That's the thing with linear models, they can't stop the growth.
- We have a negative coefficient associated with `MonthlyCharges`

In [92]:
```
cph_params.loc["MonthlyCharges"]
```

Out[92]:
```
coef   -0.003185
Name: MonthlyCharges, dtype: float64
```

If your monthly charges are huge, it takes this to the extreme and thinks you'll basically never churn.

Prediction

- We can use survival analysis to make predictions as well.
- Here is the expected number of months to churn for the first 5 customers in the test set:

In [93]:
```
test_df_surv.drop(columns=["tenure", "Churn"]).head()
```

Out[93]:

	SeniorCitizen	MonthlyCharges	TechSupport_No	TechSupport_No internet service	TechSupport_Yes	MultipleLir
941	0.0	-1.154900	1.0	0.0	0.0	
1404	0.0	-1.383246	0.0	1.0	0.0	
5515	0.0	-1.514920	0.0	1.0	0.0	
3684	0.0	0.351852	1.0	0.0	0.0	
7017	0.0	-1.471584	0.0	1.0	0.0	

5 rows × 43 columns

In [94]:
```
test_df_surv.head()
```

Out[94]:

	Churn	tenure	SeniorCitizen	MonthlyCharges	TechSupport_No	TechSupport_No internet service	TechSupport_
941	0.0	13.0	0.0	-1.154900	1.0	0.0	
1404	0.0	35.0	0.0	-1.383246	0.0	1.0	
5515	0.0	18.0	0.0	-1.514920	0.0	1.0	
3684	0.0	43.0	0.0	0.351852	1.0	0.0	
7017	0.0	51.0	0.0	-1.471584	0.0	1.0	

5 rows × 45 columns

How long each non-churned customer is likely to stay according to the model assuming that they just joined right now?

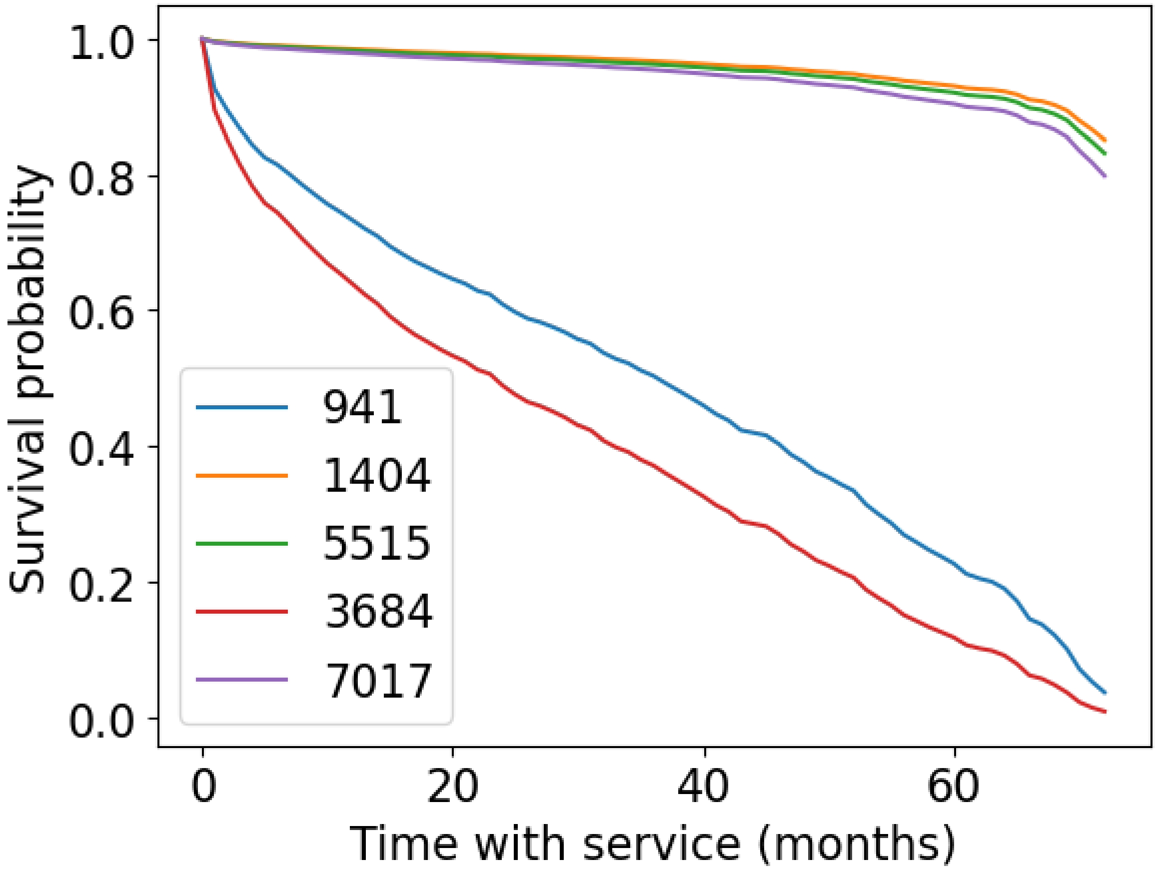
In [95]:
```
cph.predict_expectation(test_df_surv).head()  # assumes they just joine
```

Out[95]:
```
941     35.206724
1404    69.023086
5515    68.608565
3684    27.565062
7017    67.890933
dtype: float64
```

Survival curves for first 5 customers in the test set:

In [96]:

```
cph.predict_survival_function(test_df_surv[:5]).plot()
plt.xlabel("Time with service (months)")
plt.ylabel("Survival probability");
```



From `predict_survival_function` documentation:

> Predict the survival function for individuals, given their covariates. This assumes that the individual just entered the study (that is, we do not condition on how long they have already lived for.)

So these curves are "starting now".

- There's no probability prerequisite for this course, so this is optional material.
- But you can do some interesting stuff here with conditional probabilities.
- "Given that a customer has been here 5 months, what's the outlook?"
  - It will be different than for a new customer.
  - Thus, we might still want to predict for the non-churned customers in the training set!
  - Not something we really thought about with our traditional supervised learning.

Let's get the customers who have not churned yet.

In [97]:
```
train_df_surv_not_churned = train_df_surv[train_df_surv["Churn"] == 0]
```

We can *condition* on the person having been around for 20 months.

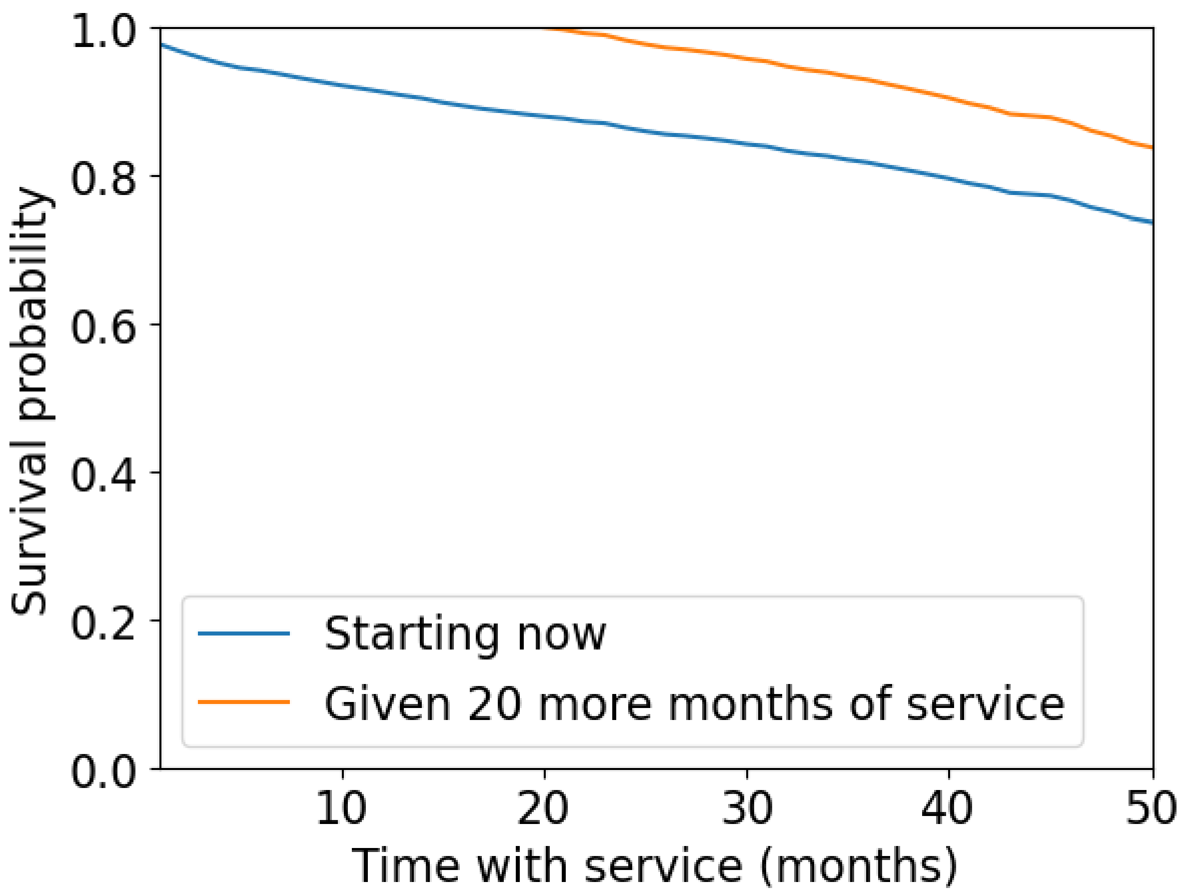
In [98]:
```
cph.predict_survival_function(train_df_surv_not_churned[:1], conditiona
```

Out[98]:

	6464
0.0	1.000000
1.0	0.996788
2.0	0.991966
3.0	0.989443
4.0	0.982570
...	...
68.0	0.429634
69.0	0.429634
70.0	0.429634
71.0	0.429634
72.0	0.429634

73 rows × 1 columns

In [99]:
```
plt.figure()
cph.predict_survival_function(train_df_surv_not_churned[:1]).plot(ax=pl
preds = cph.predict_survival_function(
    train_df_surv_not_churned[:1], conditional_after=20
)
plt.plot(preds.index[20:], preds.values[:-20])
plt.xlabel("Time with service (months)")
plt.ylabel("Survival probability")
plt.legend(["Starting now", "Given 20 more months of service"])
plt.ylim([0, 1])
plt.xlim([1, 50]);
```

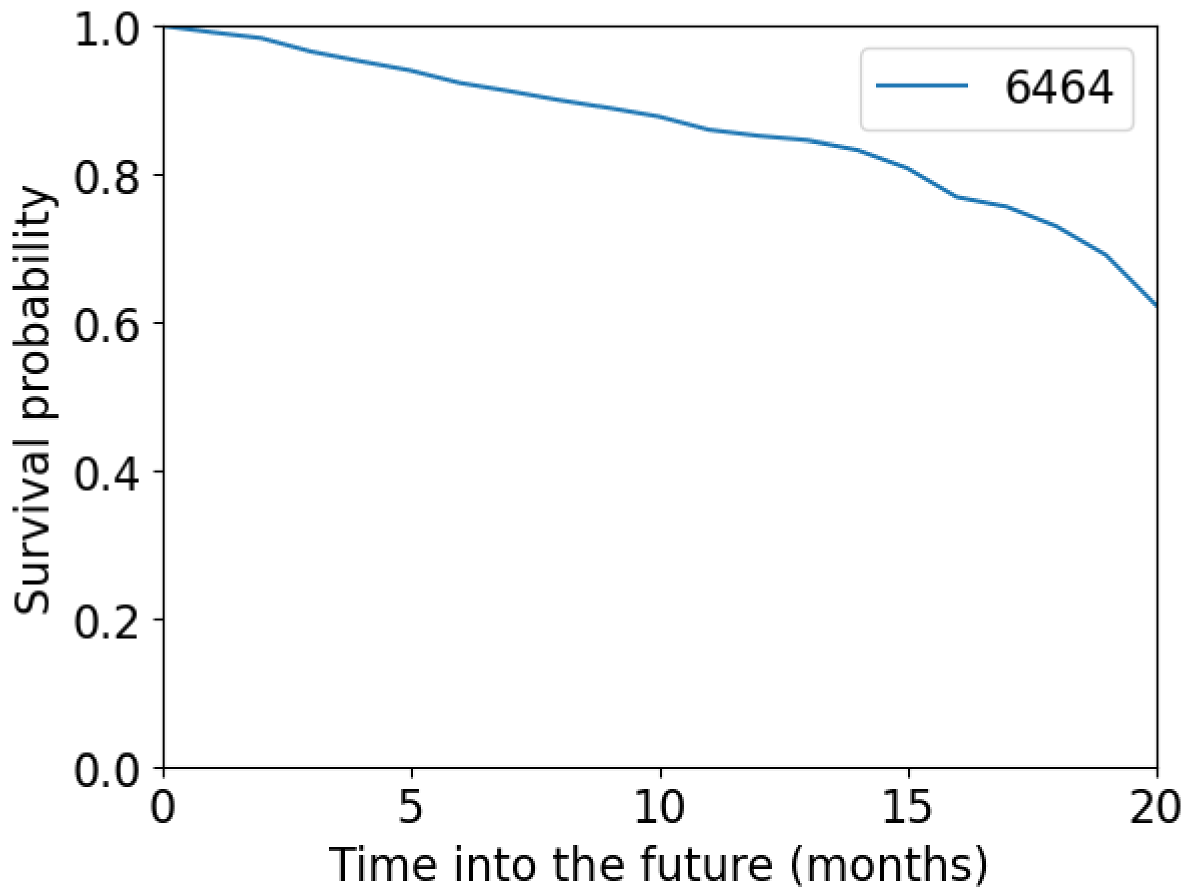


- Look at how the survival function (and expected lifetime) is much longer *given* that the customer has already lasted 20 months.

- How long each non-churned customer is likely to stay according to the model assuming that they have been here for the tenure time?
- So, we can set this to their actual tenure so far to get a prediciton of what will happen going forward:

In [100]:

```
cph.predict_survival_function(
    train_df_surv_not_churned[:1],
    conditional_after=train_df_surv_not_churned[:1]["tenure"],
).plot()
plt.xlabel("Time into the future (months)")
plt.ylabel("Survival probability")
plt.ylim([0, 1])
plt.xlim([0, 20]);
```



- Another useful application: you could ask what is the [customer lifetime value (https://en.wikipedia.org/wiki/Customer_lifetime_value)](https://en.wikipedia.org/wiki/Customer_lifetime_value).
  - Basically, how much money do you expect to make off this customer between now and when they churn?
- With regular supervised learning, tenure was a feature and we could only predict whether or not they had churned by then.

In [ ]:

```

```

Evaluation

By default score returns "partial log likelihood":

In [101]:
```
cph.score(train_df_surv)
```

Out[101]: -1.8641864337292489

In [102]:
```
cph.score(test_df_surv)
```

Out[102]: -1.7277854625841886

We can look at the "concordance index" which is more interpretable:

In [103]:
```
cph.concordance_index_
```

Out[103]: 0.8625888648969532

In [104]:
```
cph.score(train_df_surv, scoring_method="concordance_index")
```

Out[104]: 0.8625888648969532

In [105]:
```
cph.score(test_df_surv, scoring_method="concordance_index")
```

Out[105]: 0.8546143543902771

From the documentation [here (https://lifelines.readthedocs.io/en/latest/Survival%20Regression.html#model-selection-and-calibration-in-survival-regression)](https://lifelines.readthedocs.io/en/latest/Survival%20Regression.html#model-selection-and-calibration-in-survival-regression):

> Another censoring-sensitive measure is the concordance-index, also known as the c-index. This measure evaluates the accuracy of the ranking of predicted time. It is in fact a generalization of AUC, another common loss function, and is interpreted similarly:
>
> - 0.5 is the expected result from random predictions,
> - 1.0 is perfect concordance and,
> - 0.0 is perfect anti-concordance (multiply predictions with -1 to get 1.0)
>
> [Here (https://stats.stackexchange.com/a/478305/11867)](https://stats.stackexchange.com/a/478305/11867) is an excellent introduction & description of the c-index for new users.

In [96]:
```
# cph.log_likelihood_ratio_test()
```

In [97]:
```
# cph.check_assumptions(df_train_surv)
```

Other approaches / what did we not cover?

There are many other approaches to modelling in survival analysis:

- Time-varying proportional hazards.
    - What if some of the features change over time, e.g. plan type, number of lines, etc.
- Approaches based on deep learning, e.g. the [pysurvival (https://square.github.io/pysurvival/)](https://square.github.io/pysurvival/) package.
- Random survival forests.
- And more...

Types of censoring

There are also various types and sub-types of censoring we didn't cover:

- What we saw today is data with "right censoring"
- Sub-types within right censoring
    - Did everyone join at the same time?
    - Other there other reasons the data might be censored at random times, e.g. the person died?
- Left censoring
- Interval censoring

Summary

- Censoring and incorrect approaches to handling it
    - Throw away people who haven't churned
    - Assume everyone churns today
- Predicting tenure vs. churned
- Survival analysis encompasses both of these, and deals with censoring
- And it can make rich and interesting predictions!
- KM model -> doesn't look at features
- CPH model -> like linear regression, does look at the features

True/False questions

1. If all customers joined a service at the same time (hypothetically), then censoring would not be an issue.
2. The Cox proportional hazards model (`cph` above) assumes the effect of a feature is the same for all customers and over all time.
3. Survival analysis can be useful even without a "deployment" stage.

References

Some people working with this same dataset:

- https://medium.com/@zachary.james.angell/applying-survival-analysis-to-customer-churn-40b5a809b05a (https://medium.com/@zachary.james.angell/applying-survival-analysis-to-customer-churn-40b5a809b05a)
- https://towardsdatascience.com/churn-prediction-and-prevention-in-python-2d454e5fd9a5 (https://towardsdatascience.com/churn-prediction-and-prevention-in-python-2d454e5fd9a5) (Cox)
- https://towardsdatascience.com/survival-analysis-in-python-a-model-for-customer-churn-e737c5242822 (https://towardsdatascience.com/survival-analysis-in-python-a-model-for-customer-churn-e737c5242822)
- https://towardsdatascience.com/survival-analysis-intuition-implementation-in-python-504fde4fcf8e (https://towardsdatascience.com/survival-analysis-intuition-implementation-in-python-504fde4fcf8e)

lifelines documentation:

- https://lifelines.readthedocs.io/en/latest/Survival%20analysis%20with%20lifelines.html (https://lifelines.readthedocs.io/en/latest/Survival%20analysis%20with%20lifelines.html)
- https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html#introduction-to-survival-analysis (https://lifelines.readthedocs.io/en/latest/Survival%20Analysis%20intro.html#introduction-to-survival-analysis)